MASON:
A JAVA MULTI-AGENT SIMULATION LIBRARY

G. C. BALAN, Department of Computer Science
C. CIOFFI-REVILLA, Center for Social Complexity[*]
S. LUKE, Department of Computer Science [*]
L. PANAIT, Department of Computer Science
S. PAUS, Department of Computer Science
George Mason University, Fairfax, VA

ABSTRACT

Agent-based modeling (ABM) has transformed social science research by allowing researchers to replicate or generate the emergence of empirically complex social phenomena from a set of relatively simple agent-based rules at the micro-level. Swarm, RePast, Ascape, and others currently provide simulation environments for ABM social science research. After Swarm — arguably the first widely used ABM simulator employed in the social sciences — subsequent simulators have sought to enhance available simulation tools and computational capabilities by providing additional functionalities and formal modeling facilities. Here we present MASON (Multi-Agent Simulator Of Neighborhoods), following in a similar tradition that seeks to enhance the power and diversity of the available scientific toolkit in computational social science. MASON is intended to provide a core of facilities useful not only to social science but to other agent-based modeling fields such as artificial intelligence and robotics. We believe this can foster useful "cross-pollination" between such diverse disciplines, and further that MASON's additional facilities will become increasing important as social complexity simulation matures and grows into new approaches. We illustrate the new MASON simulation library with a replication of HeatBugs and a demonstration of MASON applied to two challenging case studies: ant-like foragers and micro-aerial agents. Other applications are also being developed. The HeatBugs replication and the two new applications provide an idea of MASON's potential for computational social science and artificial societies.

Keywords: MASON, agent-based modeling, multi-agent social simulation, ant foraging, aerial-vehicle flight

INTRODUCTION

Agent-based modeling (ABM) in the social sciences is a productive and innovative frontier for understanding complex social systems (Berry, Kiel, and Elliott 2002), because object-oriented programming from computer science allows social scientists to model social phenomena directly in terms of social entities and their interactions, in ways that are inaccessible either through statistical or mathematical modeling in closed form (Axelrod 1997; Axtell and Epstein 1996; Gilbert and Troitzsch 1999). The multi-agent simulation environments that have been

---

[*] *Corresponding authors' address*: Sean Luke, Department of Computer Science, George Mason University, 4400 University Drive MSN 4A5, Fairfax, VA 22030; e-mail: sean@cs.gmu.edu. Claudio Cioffi-Revilla, Center for Social Complexity, George Mason University, 4400 University Drive MSN 3F4, Fairfax, VA 22030; e-mail: ccioffi@gmu.edu. All authors are listed alphabetically.

developed in recent years are designed to meet the needs of a particular discipline; for example, simulators such as TeamBots (http://www.teambots.org) (Balch 1998) and Player/Stage (http://playerstage.sourceforge.net) (Gerkey, Vaughan, and Howard 2003) emphasize robotics, StarLogo (http://education.mit.edu/starlogo) is geared towards education, breve (Klein 2002) (http://www.spiderland.org) aims at physics and a-life, and RePast (http://www.repast.org), Ascape (http://www.brook.edu/dybdocroot/es/dynamics/models/ascape), and Swarm (http://www.swarm.org) have traditionally emphasized social complexity scenarios with discrete or network-based environments. Social science ABM applications based environments in this final category are well-documented in earlier *Proceedings* of this conference (Macal and Sallach 2000; Sallach and Wolsko 2001) and have contributed substantial new knowledge in numerous domains of the social sciences, including anthropology (hunter-gatherer societies and prehistory), economics (finance), sociology (organizations and collective behavior), political science (government and conflict), and linguistics (emergence of language) — to name a few examples.

In this paper we present MASON, a new "Multi-Agent Simulator Of Neighborhoods,, developed at George Mason University as a joint collaborative project between the Department of Computer Science's Evolutionary Computation Laboratory (Luke) and the Center for Social Complexity (Cioffi-Revilla). MASON seeks to continue the tradition of improvements and innovations initiated by Swarm, but as a more general system it can also support core simulation computations outside the human and social domain in a strict sense. More specifically, MASON is a general-purpose, single-process, discrete-event simulation library intended to support diverse multiagent models across the social and other sciences, artificial intelligence, and robotics, ranging from 3D continuous models, to social complexity networks, to discretized foraging algorithms based on evolutionary computation (EC). MASON is of special interest to the social sciences and social insect algorithm community because one of its primary design goals is to support very large numbers of agents efficiently. As such, MASON is faster than scripted systems such as StarLogo or breve, while still remaining portable and producing guaranteed replicable results. Another MASON design goal is to make it easy to build a wide variety of multi-agent simulation environments (for example, to test machine learning and artificial intelligence algorithms, or to cross-implement for validation purposes), rather than provide a *domain-specific* framework.

This paper contains three general sections. The first section describes the new MASON environment in greater detail, including its motivation, main features, and modules. The second section argues for MASON's applicability to social complexity simulation, including a comparison with RePast and a simple case-study replication of HeatBugs (a common Swarm-inspired ABM widely familiar to computational social scientists. The third section presents two further case studies of MASON applied to areas somewhat outside of the computational social science realm, but which point, we think, in directions which will be of interest to the field in the future. We conclude with a brief summary.

MASON

Motivation: Why MASON? History and Justification

MASON originated as a small library for a very wide range of multi-agent simulation needs ranging from robotics to game agents to social and physical models. The impetus for this stemmed from the needs of the original architects of the system (S. Luke , G. C. Balan, and L.
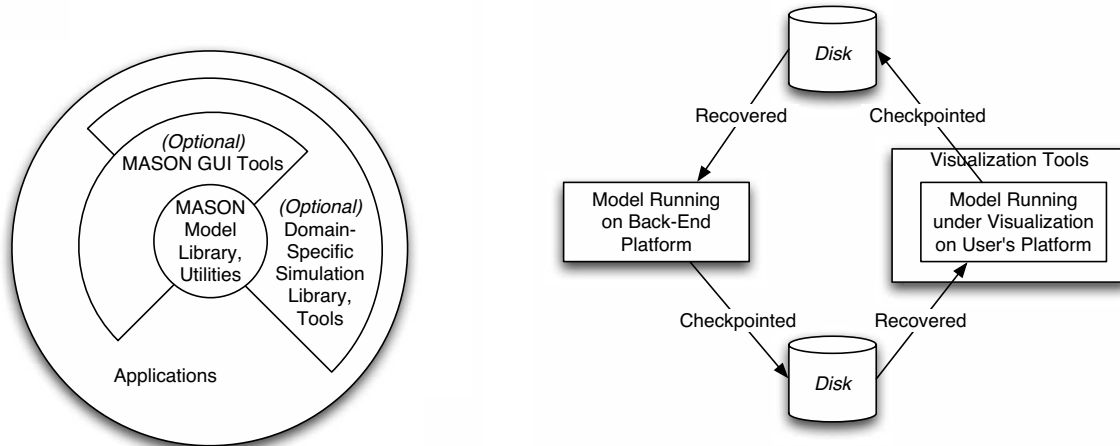
FIGURES 1 and 2 MASON Layers and Checkpointing Architecture

Panait) being computer scientists specializing in artificial intelligence, machine learning, and multi-agent behaviors. We needed a system in which to apply these methods to a wide variety of multi-agent problems. Previously various robotics and social agent simulators were used for this purpose (notably TeamBots); but domain-specific simulators tend to be complex, and can lead to unexpected bugs if modified for use in domains for which they are not designed.

Our approach in MASON is to provide the intersection of features needed for most multiagent problem domains, rather than the union of them, and to make it as easy as possible for the designer to add additional domain functionality. We think this "additive,, approach to simulation development is less prone to problems than the "subtractive,, method of modifying an existing domain-specific simulation environment. As such, MASON is intentionally simple but highly flexible.

Machine learning methods, optimization, and other techniques are also expensive, requiring a large number of simulation runs to achieve good results. Thus we needed a system that ran efficiently on back-end machines (such as beowulf clusters), while the results were visualized, often in the middle of a run, on a front-end workstation. As simulations might take a long time, we further needed built-in checkpointing to disk so we could stop a simulation at any point and restart it later.

Last, our needs tended towards parallelism in the form of many simultaneous simulation runs, rather than one large simulation spread across multiple machines. Thus MASON is a single-process library intended to run on one machine at a time.

While MASON was not conceived originally for the social agents community, we believe it will prove a useful tool for social agent simulation designers, especially as computational social science matures and grows into new approaches that require functionalities such as those implemented by the MASON environment. MASON's basic functionality has considerable overlap with Ascape and RePast partially to facilitate new applications as well as replications of earlier models in Swarm, Repast or Ascape; indeed we think that developers used to these simulators will find MASON's architecture strikingly familiar. Finally, MASON is motivated by simulation results replicability as an essential tool in advancing computationally-based claims (Cioffi-Revilla 2002), similar to the role of replication in empirical studies (Altman et al. 2001).

Features

MASON was conceived as a core library around which one might build a domain-specific custom simulation library, rather than as a full-fledged simulation environment. Such custom simulation library "flavors,, might include robotics simulation library tools, graphics and physical modeling tools, or interactive simulator environments. However, MASON provides enough simulation tools that it is quite usable as a basic "vanilla,, flavor library in and of itself; indeed, the applications described later in this paper use plain MASON rather than any particular simulator flavor wrapped around it.

In order to achieve the flavors concept, MASON is highly modular, with an explicit layered architecture: inner layers have no ties to outer layers whatsoever, and outer layers may be completely removed. In some cases, outer layers can be removed or added to the simulation dynamically during a simulation run. We envision at least five layers: a set of basic *utilities,* the *core model library*, provided *visualization toolkits*, additional custom simulation layers (flavors), and the simulation applications using the library. These layers are shown in Figure 1.

Two other MASON design goals are portability and guaranteed replicability. By replicability we mean that for a given initial setting, the system should produce identical results regardless of the platform on which it is running, and whether or not it is being visualized. We believe that replicability and portability are crucial features of a high-quality scientific simulation system because they guarantee the ability to disseminate simulation results not only in publication form but also in repeatable code form. To meet these goals, MASON is written in 100% Java.

Java's serialization facilities, and MASON's complete divorcing of model from visualization, permit it to easily perform checkpointing: at any time the model may be serialized to the disk and reloaded later. As shown in Figure 2, models may be checkpointed and loaded with or without visualization. Additionally, serialized data can be reused on any Java platform: for example, one can freely checkpoint a model from a back-end Intel platform running Linux, then load and visualize its current running state on MacOS X.

Despite its Java roots, MASON is also intended to be fast, particularly when running without visualization. The core model library encourages direct manipulation of model data, is designed to avoid thread synchronization wherever possible, has carefully tuned visualization facilities, and is built on top of a set of utility classes optimized for modern Java virtual machines.[1] Additionally, while MASON is a single-process, discrete-event library, it still permits multithreaded execution in certain circumstances, primarily to parallelize expensive operations in a given simulation.

---

[1] One efficiency optimization not settled yet is whether to use Java-standard multidimensional arrays or to use so-called ``linearized'' array classes (such as used in RePast). MASON has been implemented with both of them for testing purposes. In tight-loop microbenchmarks. linearized arrays are somewhat faster; but in full MASON simulation applications, Java arrays appear to be significantly faster. This is likely due to a loss in cache and basic-block optimization in real applications as opposed to simple microbenchmarks. We are still investigating this issue.
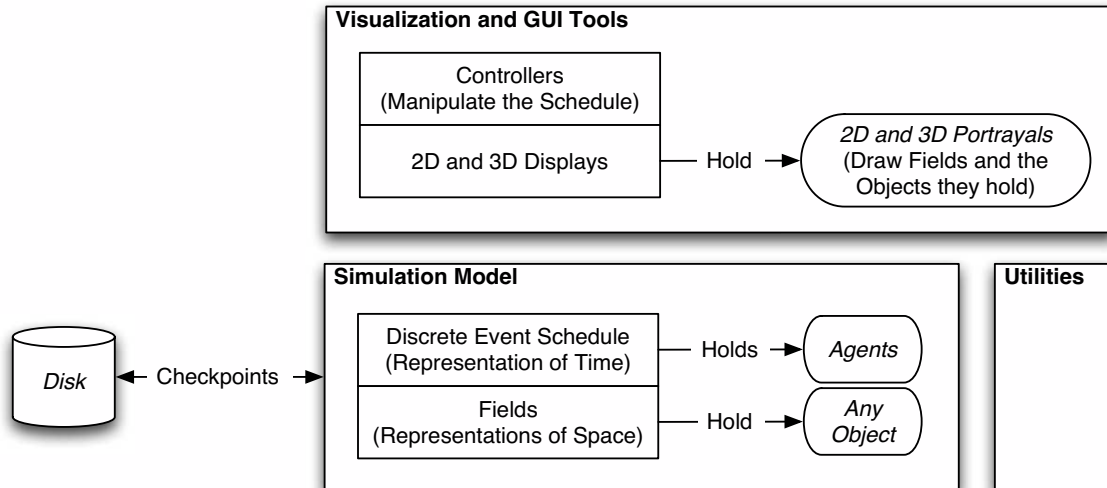
FIGURE 3 MASON Utilities, Model, and Visualization Layers

The Model and Utilities Layers

MASON's model layer, shown in Figure 3, consists of two parts: *fields* and a discrete-event *schedule*. Fields store arbitrary objects and relate them to locations in some spatial neighborhood. Objects are free to belong to multiple fields or, in some cases, the same field multiple times. The schedule represents time, and permits agents to perform actions in the future. A basic simulation model typically consists of one or more fields, a schedule, and user-defined auxiliary objects. There is some discrepancy in the use of the term *agents* between the social sciences and computer science fields. When we speak of agents, we refer to entities that can manipulate the world in some way: they are brains rather than bodies. Agents are very often *embodied* — physically located in fields along with other objects — but are not required to be so.

The model layer comes with fields providing the following spatial relationships, but others can be created easily.

- Bounded and toroidal discrete grids in 2D and in 3D for integers, doubles, and arbitrary Objects (one integer/double/Object per grid location)

- Bounded and toroidal hexagonal grids in 2D for integers, doubles, and arbitrary Objects (one integer/double/Object per grid location)

- Efficient sparse bounded, unbounded, and toroidal discrete grids in 2D and 3D (mapping zero or more Objects to a given grid location)

- Efficient sparse bounded, unbounded, and toroidal continuous space in 2D and 3D (mapping zero or more Objects to a real-valued location in space)

- Binary directed graphs or networks (a set of Objects plus an arbitrary binary relation)

The model layer does not contain any visualization or GUI code at all, and it can be run all by itself, plus certain classes in the utilities layer. The utilities layer consists of Java classes free of simulation-specific function. Such classes include *bags* (highly optimized Java Collection subclasses designed to permit direct access to int, double, and Object array data), immutable 2D and 3D vectors, and a highly efficient implementation of the Mersenne Twister random number generator.

The Visualization Layer

As noted earlier, MASON simulations may operate with or without a GUI, and can switch between the two modes in the middle of a simulation run. To achieve this, the model layer is kept completely separate from the visualization layer. When operated without a GUI the model layer runs in the main Java thread as an ordinary Java application. When run with a GUI, the model layer is kept essentially in its own "sandbox„: it runs in its own thread, with no relationship to the GUI, and can be swapped in and out at any time. Besides the checkpointing advantages described earlier, another important and desirable benefit of MASON's separation of model from visualization is that the same model objects may be visualized in radically different ways at the same time (in both 2D and 3D, for example). The visualization layer, and its relationship to the Model layer, is shown in Figure 3.

To perform the feat of separation, the GUI manages its own separate auxiliary schedule tied to the underlying schedule, to queue visualization-agents that update the GUI displays. The schedule and auxiliary schedule are stepped through a *controller* in charge of the running of the simulation. The GUI also displays and manipulates the model not directly but through *portrayals* which act as proxies for the objects and fields in the model layer. Objects in the model proper may act as their own portrayals but do not have to.

The portrayal architecture is divided into *field portrayals,* which portray fields in the model, and various *simple portrayals* stored in a field portrayal and used to portray various objects in the field portrayals' underlying field. Field portrayals are, in turn, attached to a *display*, which provides a GUI environment for them to draw and manipulate their fields and field objects. Portrayals can also provide auxiliary objects known as *inspectors* (approximately equivalent to "probes„ in RePast and Swarm) that permit the examination and manipulation of basic model data. MASON provides displays and portrayals for both 2D and 3D space, and can display all of its provided fields in 2D and 3D, including displaying certain 2D fields in 3D. 2D portrayals are displayed using AWT and Java2D graphics primitives. 3D portrayals are displayed using the Java3D scene graph library. Examples of these portrayals are shown in Figure 4.

APPLICABILITY TO SOCIAL COMPLEXITY ENVIRONMENTS

MASON was designed with an eye towards social agent models, and we think that social science experimenters will find it valuable. MASON shares many core features with social agent simulators such as Swarm, Ascape, and RePast. In this section we specify the primary differences between MASON and RePast, followed by a simple example of MASON used to simulate the well-known HeatBugs model.

a.   **b.**   **c.**

d.   **e.**   **f.**
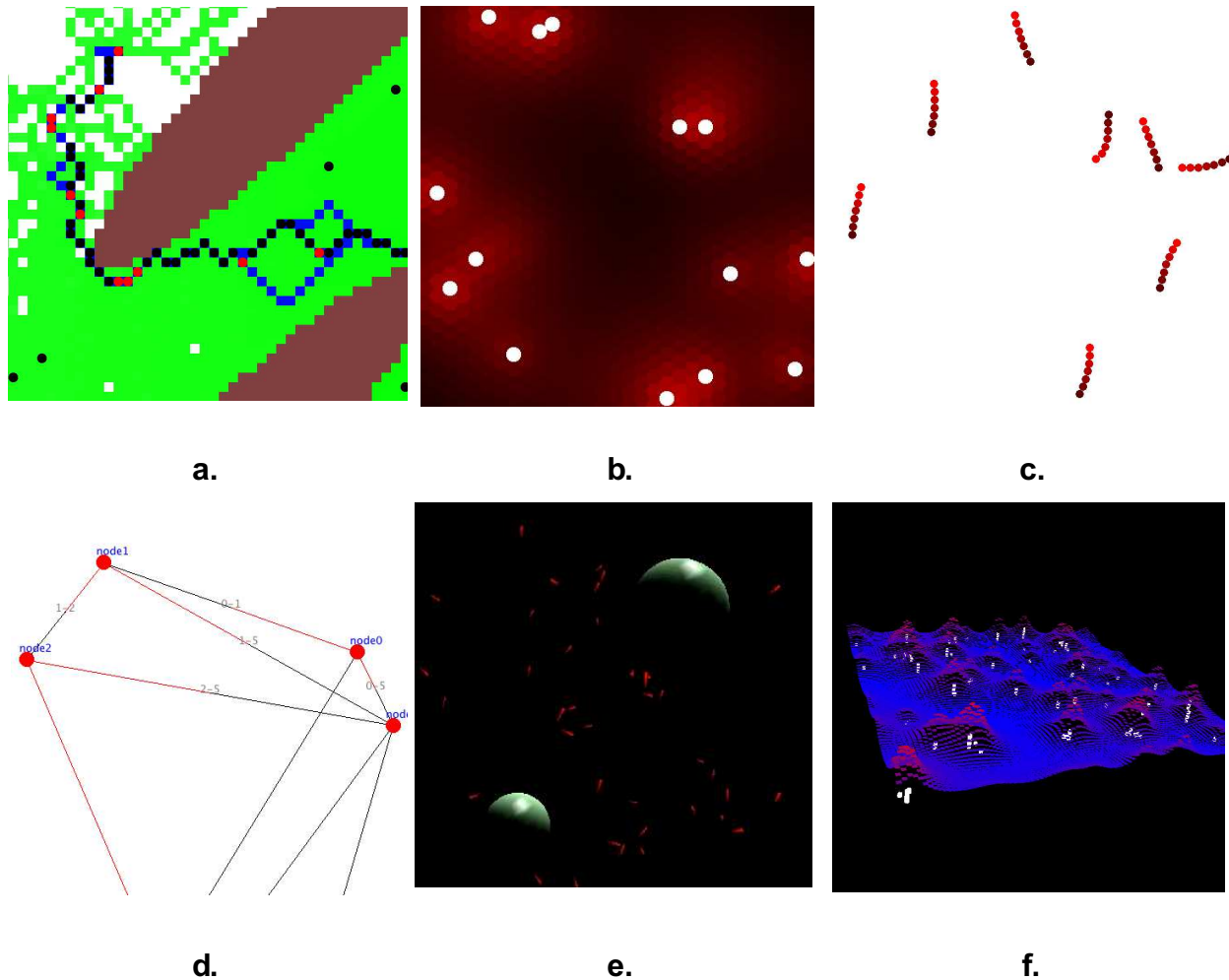
FIGURE 4 Sample Field Portrayals (Applications in Parentheses). **a.** Discrete 2D Grids (Ant Foraging). **b.** Hexagonal 2D Grids (Hexagonal HeatBugs). **c.** Continuous2D Space ("Woims": Flocking Worms). **d.** Networks in 2D (A Network Test). **e.** Continuous 3D Space (3D "Woims"). **f.** Discrete 2D Grids in 3D Space (HeatBugs)

Comparison with RePast

Here we provide a brief enumeration of most of the differences between the facilities provided by MASON and those of RePast, the latter of which evolved from Swarm to model situated social agents.

Differences

- MASON provides a full division between model and visualization. One consequence of this key difference is that MASON can separate or join the two at any time and provide cross-platform checkpointing in an easy fashion. Another consequence is that MASON objects and fields can be portrayed in very different ways at the same time, and visualization methods may change even during an expensive simulation run.

- MASON has facilities for 3D models and other visualization capabilities that remain largely unexplored in the social science realm, but which are potentially insightful for social science ABM simulations.

- In our experience, MASON has generally faster models and visualization than RePast, especially on Mac OS X, and has more memory-efficient sparse and continuous fields. MASON's model data structures also have computational complexity advantages.

- MASON has a clean, unified way of handling network and continuous field visualization.

- RePast provides many facilities, notably: GIS, Excel import/export, charts and graphs, and SimBuilder and related tools. Due to its design philosophy, MASON does not include these facilities. We believe they are better provided as separate packages rather than bundled. Further, we believe that many of these tools can be easily ported to MASON.

Differences in Flux: MASON Will Likely Change These Features in its Final Version

- RePast uses linearized array classes for multidimensional arrays. MASON presently has facilities for both linearized arrays and true Java arrays, but may reduce to using one or the other.

- RePast's schedule uses doubles, while MASON's schedule presently uses longs.

- RePast allows objects to be selected and moved by the mouse.

- RePast allows deep-inspection of objects; MASON's inspection is presently shallow.

Replicating HeatBugs

HeatBugs is arguably the best known ABM simulation introduced by Swarm, and is a standard demonstration application in RePast as well. It contains basic features common to a great many social agent simulations: for example, a discrete environment defining neighborhood relationships among agents, residual effects (heat) of agents, and interactions among them. We feel that the ability to replicate models like HeatBugs, Sugarscape, Conway's Game of Life (or other cellular automata), and Schelling's segregation model in a new computational ABM environment should be as essential as the ability to implement regression, factor analysis, ANOVA, and similar basic data facilities in a statistical analysis environment.

Indeed, a 100x100 toroidal world, 100-agent HeatBugs model was MASON's very first application. In addition to this classical HeatBugs model, we have implemented several other HeatBugs examples. Figure 4 includes partial screenshots of two of them: Figure 4B shows HeatBugs on a hexagonal grid (fittingly called "HexaBugs,, in RePast), and Figure 4f shows 2D HeatBugs visualized in 3D space, where vertical scale indicates temperature, and HeatBugs on

the same square are shown stacked vertically as well. Whereas the original HeatBugs is based on a 2D grid of interacting square cells (connected by Moore or von Neumann neighborhoods), HexaBugs is more relevant in some areas of computational social science where hexagonal cells are more natural (e.g., computational political science, especially international relations) and four-corner situations are rare or nonexistent (Cioffi-Revilla and Gotts 2003).

CASE STUDIES

Although MASON has existed for only six months, we have already used it in a variety of research and educational contexts. Additionally, we are conducing tests to port RePast, Swarm, and Ascape models to MASON, ported by modelers not immediately familiar with MASON. These ports include a model of warfare among countries, a model of land use in a geographic region, and a model of the spread of anthrax in the human body).

In this section we describe the implementation and results of two research projects that used the MASON simulation library. The first case study used MASON to discover new ant-colony foraging and optimization algorithms. The second case study applied MASON to the development of evolved micro-aerial vehicle flight behaviors. These are not computational social science models per se: but they are relevant enough to prove illuminating. The first case study uses a model that is similar to the discrete ABM models presently used, but it is applied to an automated learning method, demonstrating the automated application of large numbers of simulations in parallel. The second case study uses a continuous 2D domain environment and interaction, which we think points to one future area of ABM research. Neither of these more advanced applications is implemented in Swarm, RePast, or Ascape at present, and both take advantage of features special to MASON. In both cases experiments were conducted running MASON on the command-line in several back-end machines, and the progress was analyzed by attaching the simulators to visualization tools on a front-end workstation. Additionally, the second case involves a continuous field that is scalable and both memory- and time-efficient (both O(#agents), rather than O(spatial area)).

Both of the projects described below have an evolutionary computation (EC) component: to save repetition, we give a quick explanation of evolutionary computation here. EC is family of stochastic search and optimization techniques for "hard„ problems for which there is no known procedural optimization or solution-discovery method. EC is of special interest to certain multiagent fields because it is *agent-oriented*: it operates not by modifying a single candidate solution, but by testing a "population„ of such solutions all at one time. Such candidate solutions are known as "individuals„ and each individual's assessed quality is known as its "fitness„. The general EC algorithm is as follows. First, an initial population of randomly-generated individuals is created and each individual's fitness is assessed. Then a new population of individuals (the next generation) is assembled through an iterative process of stochastically selecting individuals (tending to select the fitter ones), copying them, then breeding the copies (mixing and matching individuals' components and mutating them), and placing the results into the next generation. The new generation replaces the old generation; its individuals' fitnesses are in turn assessed, and the cycle continues. EC ends when a sufficiently fit individual is discovered, or when resources (notably time) expire. The most famous example of EC is the *genetic algorithm* (GA) (Holland 1975), but other versions exist as well; we will discuss *genetic programming* (GP) (Koza 1992) as one alternative EC method below.

Ant Foraging

Ant foraging models attempt to explain how ant colonies discover food sources, then communicate those discoveries to other ants through the use of pheromone trails — leaving proverbial "bread crumbs„ to mark the way. This area has become popular not just in biology but curiously in artificial intelligence and machine learning as well, because pheromone-based communication has proven an effective abstract notion for new optimization algorithms (known collectively as *ant colony optimization*) and for cooperative robotics.

Previous ant foraging models have to date relied to some degree on *a priori* knowledge of the environment, in the form of explicit gradients generated by the nest, by hard-coding the nest location in an easily-discoverable place, or by imbuing the ants with the knowledge of the nest direction. In contrast, the case study presented here solves ant foraging problems using two pheromones, one applied when leaving the nest and one applied when returning to the nest. The resulting algorithm is orthogonal and simple, and biologically plausible, yet ants are able to establish increasingly efficient trails from the nest to the food even in the presence of obstacles.

Ants are sensitive to one of the two pheromones at any given time; the sensitivity depends on whether they are foraging or carrying food. While foraging, an ant will stochastically move in the direction of increasing food pheromone concentration, and will deposit some amount of nest pheromone. If there is already more nest pheromone than the desired level, the ant deposits nothing. Otherwise, the ant "tops off„ the pheromone value in the area to the desired level. As the ant wanders from the nest, its desired level of nest pheromone drops. This decrease in deposited pheromone establishes an effective gradient. When the ant is instead carrying food, the movement and pheromone-laying procedures use the opposite pheromones than those used during foraging.

The model assumes a maximum number of ants per location in space. At each time step, an ant will move to its best choice among non-full, non-obstacle locations; the decision is made stochastically with probabilities correlated to the amounts of pheromones in the nearby locations. Ants move in random order. Ants live for 500 time steps; a new ant is born at the nest each time step unless the total number of ants is at its limit. Pheromones both evaporate and diffuse in the environment.

Figure 4a shows a partial screenshot of a small portion of the ant colony foraging environment. The ants have laid down a path from the nest to the food and back again. Part of the ground is colored with pheromones. The large oval regions are obstacles. The MASON implementation was done with two discrete grids of doubles (two pheromone values), discrete grids of obstacles, food sources, and ant nests, and a sparse discrete grid holding the ants proper. Each ant is also an agent (and so is scheduled at each time step to move itself). Additional agents are responsible for the evaporation and diffusion of pheromones in the environment, and also for creating new ants when necessary.

In addition to the successful design of hard-coded ant foraging behaviors, we also experimented with letting the computer search for and optimize those behaviors on its own. For this purpose, we connected MASON to the ECJ evolutionary computation system (Luke 2002) (http://cs.gmu.edu/~eclab). ECJ handled the main evolutionary loop: an individual took the form of a set of ant behaviors that was applied to each ant in the colony. To evaluate an individual, ECJ spawned a MASON simulation with the specified ant behaviors. The simulation was run

for several hundred timesteps. At the end of the simulation, the amount of food foraged indicated the individual's fitness.

To evolve ant behaviors, we used *genetic programming* (GP) (Koza 1992). In genetic programming, individuals are actual computer programs in the form of one or more *parse trees*. We will not describe parse trees except to explain that breeding consisted of swapping subtrees among individuals. Our EC individuals (the behaviors) consisted of two such GP tree-programs: the execution of one tree returned the amount of pheromone to deposit, and the execution of the other tree yielded the direction to move. The same behavior was used in both an ant's foraging and food-carrying states; but the pheromones specified in its behaviors were swapped (food pheromone for nest pheromone, and vice versa).

A first experiment scaled the number of ants (50, 50, 500), the number of simulation time steps (501, 1001, 2501), and the world size (10x10, 33x33, 100x100). In each case the EC populations converged rapidly to simple but reasonably high-performing ant foraging behaviors. Increasing the world size led to longer convergence times (from a mere two generations in the 10x10 case to ten generations on average in the 100x100 case).

Interestingly, these behaviors were different in meaningful ways from one another. This led to a comparison of the three highest-performing behaviors obtained with the various settings previously mentioned. The results showed that more difficult domains led to the discovery of more robust foraging strategies. Additional details on this work can be found in (Panait and Luke 2003a; Panait and Luke 2002b).
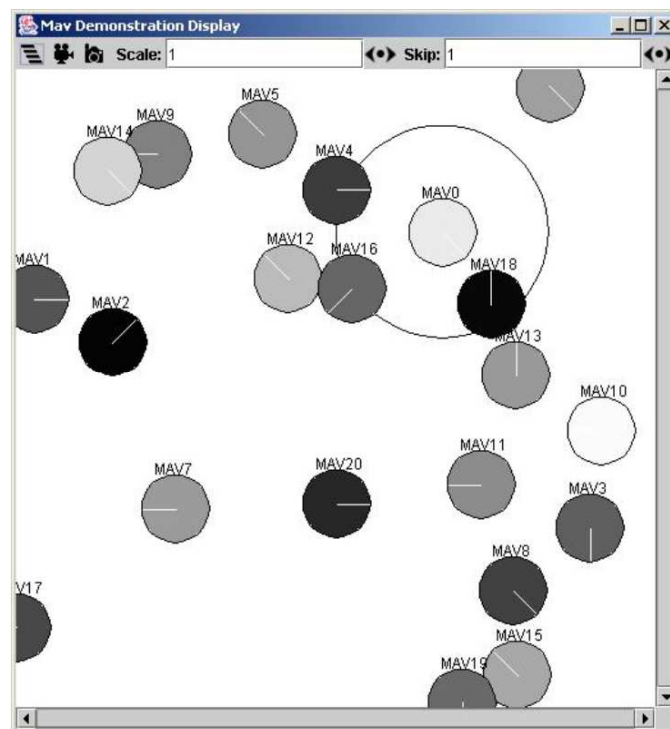
FIGURE 5 Micro-Aerial Vehicle Simulation in MASON. Vehicles (the circles) appear much larger than they actually are. Gray values indicate level of dominance, and lines indicate orientation of each vehicle.

Micro-Aerial Vehicle Simulation

An *Unmanned Aerial Vehicle* (UAV) is a flying device, often an airplane, operated by remote control usually for military functions (such as surveillance, reconnaissance, or attack). The UAV most familiar to the general public is the Predator, a flying drone by General Atomics that has flown surveillance missions over Afghanistan and Iraq. Large UAVs such as the Predator are expensive to produce; moreover even though they have no on-board pilot, UAVs require a large team of controllers on the ground to fly the vehicle. One recent thrust in UAVs has been the *Micro-Aerial Vehicle* (MAV), a tiny (less than 1 meter), inexpensive UAV meant primarily for surveillance. Being cheap, MAVs are often designed to fly in "swarms,, of up to hundreds of vehicles; such swarms mean that a unique human controller cannot be feasibly allocated for each MAV. Instead, it is hoped that an entire MAV swarm may be controlled by a single, small team of controllers. To achieve this, MAVs must be *semi-autonomous*: they receive high-level commands from human controllers, but most of the dirty work is achieved by the MAVs all by themselves.

The University of Central Florida and George Mason University have recently worked on a joint DARPA project demonstrating the feasibility of having swarms of MAVs learn behaviors in simulation via evolutionary computation. The research system developed was a combination of an evolutionary computation system developed at UCF, a library of *dominance hierarchies* developed at UCF, and a MASON simulation environment. The system was recently completed and published results are forthcoming.

The MASON simulation held MAVs in a continuous 2D neighborhood along with *regions*, colored shapes "painted,, on the ground, over which the MAVs would fly. Our goal was to develop MAV behaviors which caused them to fly over specific colored regions as much and as often as possible while avoiding collisions with one another. If MAVs collided, they were removed from the simulation.

Our MAV behaviors consisted of sets of basic *sensorValues→action* rules: a rule might say, for example, that if the MAV was directly above the appropriate region color, and there was another nearby MAV to the upper-left, then the MAV should turn right. An additional "sensor,, available to the rules was the *dominance* of the MAV relative to its neighbors; nearby MAVs established dominance hierarchies amongst themselves using a method developed by Tomlinson (2002).

After some number of timesteps, the MAV swarm's quality was assessed by adding up the total time each MAV was located over a region of interest. We applied an evolutionary computation system to learn MAV behaviors that, when used by a MAV swarm in the simulator, produced as high-quality assessments as possible.

The system was constructed in MASON as follows. Each MAV was an embodied MASON agent, and was also stored in a continuous 2D field. Regions were stored in a second continuous 2D field. Each MAV held a ring of eight "sonar sensors,, (rays emanating in eight directions from the MAV). At each timestep, each MAV would call the provided dominance library to update its dominance values based on the relative values of nearby MAVs. It then would determine the distance to the closest MAV that intersected each sonar ray. These eight distance values, plus the value indicating the color of the region presently below the MAV, plus

the current dominance value of the MAV, formed the MAV's ten sensor values. The MAV then determined which rule in its ruleset most closely matched its current sensor values, and performed that action. An action consisted of one of eight directions in which the MAV could turn. After turning in that direction, the MAV would then move forward some distance. If it then collided with other MAVs, they would be all eliminated from the simulation.

Once again, MASON was used as a subsidiary process to an evolutionary computation system, this time one devised by Prof. Annie Wu at the University of Central Florida. The individuals (the MAV behaviors) were represented in a genetic algorithm as vectors of numbers indicating the direction to fly given various sensor values. An individuals' fitness was assessed by creating a MAV simulation in MASON, plugging the behavior into the MAVs in the model, running the model for some N time steps, then assessing the total amount of time MAVs stayed over appropriate target regions.

SUMMARY

Agent-based modeling (ABM) has already begun to transform social science research — "the third way of doing science,, (Axelrod 1997) — by allowing researchers to replicate or generate the emergence of empirically complex social phenomena from a set of relatively simple agent-based rules at the micro-level. One of the keys to this transformation has been object-oriented modeling (Gulyás 2002), which moves beyond models in closed form (Taber and Timpone 1996). Swarm, RePast, Ascape, and other simulation environments already provide numerous capabilities for ABM social science research. After Swarm — arguably the first widely utilized ABM simulator employed in the social sciences — subsequent simulators have sought to enhance available simulation tools and computational capabilities by providing additional functionalities and formal modeling facilities. Here we presented MASON (Multi-Agent Simulator Of Neighborhoods), which follows in a similar tradition that seeks to enhance the power and diversity of the available scientific toolkit in computational social science, artificial intelligence, and other multi-agent areas. We argue that besides its immediate use in conducting social complexity simulations, MASON provides a general framework to serve as a core for a wide range of multi-agent needs, many of which will become increasingly important as social complexity simulation matures and grows into new approaches. We illustrated the new MASON simulation library with a replication of HeatBugs and a demonstration of two challenging MASON applications as case studies: ant-like foragers and micro-aerial vehicles. Other applications are also being developed to demonstrate and enhance MASON's features. The HeatBugs replication and the two new applications provide an idea of MASON's potential for computational social science and artificial societies.

ACKNOWLEDGEMENTS

REFERENCES

Altman, Micah, L. Andreev, M. Diggory, Gary King, Sidney Verba, Daniel L. Kiskis, and M. Krot. 2001. A digital library for the dissemination and replication of quantitative social science research: The Virtual Data Center. *Social Science Computer Review* 19 (4).

Axelrod, Robert. 1997. Advancing the art of simulation in the social sciences. *Complexity* 3 (2):193-99.

Axtell, Robert, and Joshua M. Epstein. 1996. *Growing Artificial Societies: Social Science From the Bottom Up*. Cambridge, MA: MIT Press.

Balch, Tucker. 1998. *Behavioral Diversity in Learning Robot Teams*. PhD Dissertation. College of Computing, Georgia Institute of Technology.

Berry, Brian L., L. Douglas Kiel, and Euel Elliott, eds. 2002. *Adaptive Agents, Intelligence and Emergent Human Organization: Capturing Complexity Through Agent-Based Modeling*. Vol. 99, *Proceedings of the National Academy of Sciences*: National Academy of Sciences.

Cioffi-Revilla, Claudio. 2002. Invariance and universality in social agent-based simulations. *Proceedings of the National Academy of Science of the U.S.A.* 99 (Supp. 3) (14):7314-7316.

Cioffi-Revilla, Claudio, and Nicholas M. Gotts. 2003. Comparative analysis of agent-based social simulations: GeoSim and FEARLUS models. *Journal of Artificial Societies and Social Systems* 6 (4).

Gerkey, Brian, Richard T. Vaughan, and Andrew Howard. 2003. "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems" *Proceedings of the 11th International Conference on Advanced Robotics*, 317–323.

Gilbert, Nigel, and Klaus Troitzsch, eds. 1999. *Simulation for the Social Scientist*. Buckingham and Philadelphia: Open University Press.

Gulyás, Lászlo. 2002. On the transition to agent-based modeling: Implementation strategies from variables to agents. *Social Science Computer Review* 20 (4):389-399.

Holland, John. 1975. *Adaptation in Natural and Artificial Systems*. MIT Press.

Klein, J. 2002. breve: a 3D simulation environment for the simulation of decentralized systems and artificial life. *Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems* . MIT Press.

Koza, J. 2002. *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. MIT Press.

Luke, Sean. 2000. *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*. PhD Dissertation. Department of Computer Science, University of Maryland, College Park.

Macal, Charles M., and David Sallach, eds. 2000. *Proceedings of the Workshop on Agent Simulation: Applications, Models, and Tools*. Chicago: Social Science Research Computation, The University of Chicago, and Decision & Information Sciences Division, Argonne National Laboratory.

Panait, Liviu, and Sean Luke. 2003a. "Ant Foraging Revisited" *Second International Workshop on the Mathematics and Algorithms of Social Insects*. (Submitted)

Panait, Liviu, and Sean Luke. 2003b. "Evolving Foraging Behaviors" *Second International Workshop on the Mathematics and Algorithms of Social Insects*. (Submitted)

Sallach, David, and Thomas Wolsko, eds. 2001. *Proceedings of the Workshop on Simulation of Social Agents: Architectures and Institutions*. Chicago, IL: Social Science Research Computation, The University of Chicago, and Decision and Information Sciences Division, Argonne National Laboratory.

Taber, Charles S., and Richard J. Timpone. 1996. *Computational Modeling*. Edited by M. S. Lewis-Beck. Vol. 07-113, *Sage University Papers, Quantitative Applications in the Social Sciences*. Thousand Oaks, London and New Delhi: Sage Publications.

Tomlinson, William. 2002. *Synthetic Social Relationships for Computational Entities*. PhD Dissertation. Program in Media Arts & Sciences. MIT. Available at http://web.media.mit.edu/~badger/pubs.html